From concurrent multi-clock programs to concurrent multi-threaded implementations

Virginia Papailiopoulou, Dumitru Potop-Butucaru, Yves Sorel, Robert de Simone, Loic Besnard, Jean-Pierre Talpin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Synthèse d'implantations multi-thread concurrentes à partir de programmes multi-horloges

Virginia Papailiopoulou — Dumitru Potop-Butucaru — Yves Sorel — Robert de Simone — Loïc Besnard — Jean-Pierre Talpin

N° 7577 — version 2

version initiale Mars 2011 — version révisée Juillet 2011

Embedded and Real Time Systems

apport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7577--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Synthèse d'implantations multi-thread concurrentes à partir de programmes multi-horloges

Virginia Papailiopoulou , Dumitru Potop-Butucaru , Yves Sorel ,
Robert de Simone* , Loïc Besnard† , Jean-Pierre Talpin†

Theme : Embedded and Real Time Systems
Équipes-Projets Aoste et Espresso

Rapport de recherche n° 7577 — version 2 — version initiale Mars 2011 —
version révisée Juillet 2011 — 22 pages

Résumé : On présente dans ce rapport une nouvelle technique pour l'implantation concurrente de spécifications multi-horloges. A partir de programmes écrits en langages synchrones comme Signal/Polychrony ou Esterel, on fournit des structures de données compactes et des algorithmes efficaces pour l'analyse du programme selon la théorie des systèmes faiblement endochrones (weakly endochronous systems). Les exécutifs multi-thread qui sont générés par la technique proposée produisent des implantations déterministes en conservant la concurrence de la spécification originale.

Ce travail a été partiellement financé par le projet HELP de l'Agence Nationale de la Recherche (ANR-09-SEGI-006).

Mots-clés : endochronie, specification synchrone, multi-horloges, implantation asynchrone

* UR de Sophia Antipolis Méditerranée
† IRISA Rennes

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

From concurrent multi-clock programs to concurrent multi-threaded implementations

Abstract: We present a new technique for the concurrent asynchronous/GALS implementation of polychronous specifications. We start from programs written in multi-clock languages such as Signal/Polychrony or Esterel. We provide compact data structures and corresponding algorithms for program analysis, following the theory of weakly endochronous systems. Finally, we produce multi-threaded deterministic asynchronous/GALS implementations that retain as much as possible of the concurrency of the initial specification.

This paper has been partially supported by the French ANR project HELP (ANR-09-SEGI-006).

Key-words: endochrony, synchrony, multi-clock, asynchronous implementation

1 Introduction

The synchronous approach is widely used nowadays for the design of safety-critical applications, such as digital circuits or embedded software. The well defined notions of time and causality at the specification level provide a simple way to model, analyze and verify a system. Deriving a correct physical implementation of the synchronous model must be done in such a way that determinism and functional correctness are preserved. However, this is not always guaranteed mainly because of the distributed nature of the implementation platform which does not support the synchrony hypothesis. In addition, computation and communication speeds vary according to the implementation platform, making it difficult to maintain a global notion of time. Another major problem is that the absence of an event, which is well defined in the synchronous model, does not truly make sense in an asynchronous setting (it is time-dependent).

The classical brute-force way of implementing synchronous processes in an (asynchronous) distributed setting consists in broadcasting the full sequence of signal values, including an explicit *absent* one to represent those instants in the synchronous versions where the signal is indeed absent. This sequence, used wherever the signal is watched for, recreates in essence a distributed version of the synchronous instants. It is easily seen that such a solution requires a huge constant number of actual signaling (by which absence information is transmitted). A large body of theoretical studies have been conducted so far on ways to decrease this traffic of absence notifications. They consider the fact that, below the original synchronous surface, some of the behaviors were actually already independent (and mutually asynchronous, with disjoint parts reacting on disjoint inputs), so they can safely be kept asynchronous without involving the common instant recreation induced by these absence notifications. Of course detailed treatment here may become much more hairy, and our current work consists in part of an atempt to improve on existing techniques, which we shall broadly qualify as *endochrony checking*.

We address this issue, considering implementation platforms that support lossless order-preserving asynchronous message passing over the communication channels. The objective is to preserve the functional properties of synchronous programs when the latter are implemented on an asynchronous environment. To this aim, our approach focuses on a multi-clock model of computation and defines proper algorithms in order to determine and ensure that the original program can be safely executed over an asynchronous architecture with identical results as expected by the specifications.

The formal theoretical background of the proposed technique is that of weakly endochronous systems [15]. Weak endochrony defines the necessary and sufficient conditions that a synchronous program must satisfy in order to ensure the notions of confluence and monotony. Informally, this means that a synchronous process is weakly endochronous if information about the absence of a signal is not needed in computations. As a consequence, program transitions are distinguished by different signal values and not by the presence or absence of a signal, ensuring the functional correctness of the implementation. Therefore, weak endochrony provides a latency-insensitivity and scheduling-independence criterion in the context of synchronous systems.

A general description of our method is shown in Fig. 1. Starting from a multi-clock concurrent program, we compute a minimal set of synchronization

patterns, the so-called *generator sets*. Generator sets provide a compact representation of any synchronous program, describing its behavior using as few synchronization information as possible for its correct implementation. If explicit absence information is not needed to distinguish between any two elements of a generator set, then the original program is weakly endochronous and can be deployed on an asynchronous execution environment. In the opposite case, the computed generator set points directly to the defective synchronization pattern, providing the solution for rendering the program weakly endochronous. At this point, the programmer can either accept the proposed solution or proceed with his own modifications on the original program.

Contribution. The current paper builds up on a series of articles by us and fellow co-authors, from the early motivations of [2] to extensive developments of the theory behind weak endochrony as an important methodological tool for asynchronous distribution of synchronous programs [15, 10, 17]. Here we focus on practical algorithmic aspects and efficient symbolic data structure representations. We show how to compute the minimal sets of mutually independent behavior generators, and how to encode them into efficient multi-threaded code. We have implemented our symbolic data structures and algorithms in a prototype version, linked to the Signal/Polychrony-SME design environment as a back-end tool. Thus, programs are written in Signal, and various other analysis available for this language are thus possibly combined with our distributed code generation. Benchmarking on existing large Signal programs is under way.

Outline. The remainder of the paper is organized as follows : Section 2 provides an intuitive, example-based description of our code generation problem and reviews the previous and related work. The presentation is rather long, but each example will be needed later in the paper to illustrate an analysis or code generation problem. Section 3 briefly reviews the theory of weakly endochronous systems, which founds our solution. Section 4 explains how efficient multi-threaded code is generated from the atom set of a weakly endochronous program. Much emphasis is put here on the symbolic and hierarchical representation of the atom set, which impacts the quality of the generated code. Section 5 explains how the symbolic representation techniques are extended to generator sets used by the endochrony analysis algorithms. We conclude in Section 7.

2 Problem description

The synchronous programming paradigm is now well established, and made popular because of its role at the joint point of 1) computer science and language design, 2) control theory and reactive systems, and 3) microelectronic (synchronous) circuit design. It provides a sound semantic background with a notion of discrete instants and successive reactions, together with high-level structuring primitives which help define subthreads whose activations (defined by signals or clocks) model over- and sub-sampling.

This "modelling comfort" of synchronous programs, where the control over the content of the activation step is available to the programmer for design decisions, comes at a cost. Semantics is sound, clear and formal, but the results of compilation are quite heavy : Compilation in itself is a complex process as it includes a check of proper causality and an expansion into full-fledged flat structures such as Mealy FSMs or boolean equation systems (digital netlists).

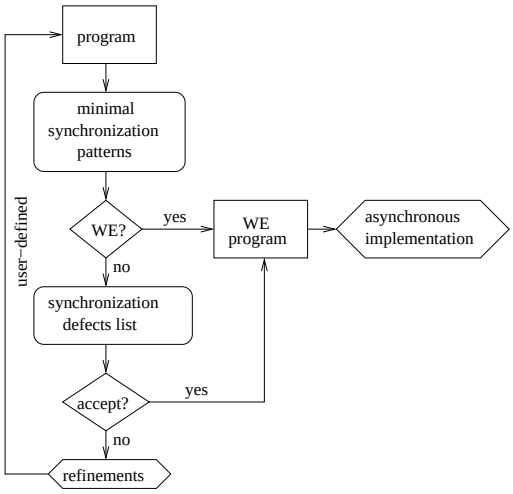
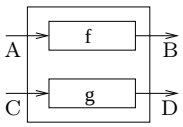


FIG. 1 – General description of the proposed method




```
process SIMPLE =
(? int A,C; ! int B,D;)
(| B := f(A)
 | D := g(C) |) end
```

```
module SIMPLE:
input  A,C:int ; output B,D:int ;
[
  every A do emit B(f(?A)) end
||
  every C do emit D(g(?C)) end
]
end module
```

FIG. 2 – A simple system formed of two independent sub-systems. Global data-flow (top left), Esterel code (bottom), and Signal/Polychrony code (top right).

Most of all, while an asynchronous/causal style of programming is usually proposed (various primitives can start/stop/pause a task upon a signal/subclock condition), this is **not** exploited in the course of this semantic expansion at compile time, which provides a flat expanded version of the design specification, running on a single base clock (the *reaction step*).

Exploiting the semantic independence of various computations to allow the generation of concurrent, potentially distributed code from synchronous and polychronous specifications is a notoriously difficult subject. It amounts to determining which part of the system-wide synchronization specific to the synchronous model can be removed while preserving the specified functionality. We illustrate the basic implementation problem with the very simple example of Fig. 2, which represents a synchronous program encapsulating two independent

computations : One which produces values (messages) over output B in response to input messages on A, and one which produces messages over D in response to messages on C. For every message x received on A, the value f(x) is produced on B, and for every message y received on C, the message g(y) is produced on D.

```
bool A_present,C_present; int A,C;
void reaction_function() {
  if(A_present) emit_B(f(A)) ;
  if(C_present) emit_D(g(C)) ;
}
void main() { /* driver routine */
  for(;;) {
    await_trigger() ;
    reaction_function() ;
  } }
```

FIG. 3 – Single-loop implementation of the SIMPLE program.

The classical single-loop implementation produced for this example by existing synchronous language compilers has the structure of Fig. 3. Computations of reaction_function are triggered cyclically by the driver (periodically, or on arrival of input messages, depending on the implementation). The function checks whether A and C have arrived, and emits B and D accordingly. Distributing such a globally synchronized implementation is difficult, because it involves detecting that the computations of the two if statements are independent.

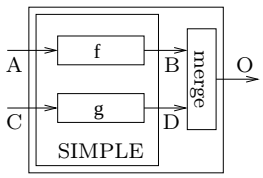
The code we want to generate, presented in Fig. 4, has two threads. The first cyclically executes the function onA, which waits for messages on A and produces the corresponding outputs on B. The second thread waits for messages on C and produces outputs on D. The two computations being fully independent, this coding is natural and allows (if needed) a simple distribution without synchronization overheads.

```
void onA () { int A ;
  await_A(&A) ; emit_B(f(A));
}
void onC () { int C ;
  await_C(&C) ; emit_D(g(C));
}
void main() { /* driver pseudocode */
  start_thread{ for(;;) onA() ; } ;
  start_thread{ for(;;) onC() ; } ; }
}
```

FIG. 4 – Concurrent, event-driven implementation of SIMPLE.

One could argue that this example is best specified, analyzed, and implemented as the asynchronous composition of two independent computations. This is true when considered in isolation. However, current real-life avionics, rail, and other embedded systems involve a plethora of sub-systems, at a multitude of abstraction/integration levels. Many of these are naturally modeled as synchro-

nous, which justifies the use of a global synchronous modeling framework and makes the code generation problem mentioned above meaningful.




```
process MERGE =
(? int A,C; ! int O;)
(| (B,D) = SIMPLE(A,C)
 | O = B default D
 | A ^# C |)
where int B,D; end
```

```
void reaction_function() {
  if(A_present) emit_O(f(A)) ;
  else if(C_present) emit_O(g(C)) ;
}
```

FIG. 5 – Extending `SIMPLE` with a shared resource. Dataflow (top left), Signal code (top right), single-loop implementation (bottom). The driver is that of Fig. 3.

Of course, a new code generation technique exploiting concurrency should not penalize the (non-concurrent) cases where classical single-loop code generation works well. We give several more examples to better illustrate our objective.

The simplest way of breaking the concurrency of a specification like `SIMPLE` is by introducing a shared resource under the form of a code fragment needed by both computations. We give an example of this in Fig. 5, by merging into a single stream, named `O`, the outputs produced on `B` and `D`. To avoid collisions, we require that inputs never arrive on both `A` and `C` in a given execution cycle. We say that `A` and `C` are *exclusive*, which is represented using operator `#` in both Esterel and Signal.

Implementations of `MERGE` are easily produced through simple modifications of the corresponding implementations of `SIMPLE`. However, the behavior of these implementations depends not only on the streams of values received on `A` and `C`, but also their relative arrival order. In turn, the arrival order depends on many implementation details like the specific I/O mechanisms, the distribution, the computation and communication speeds, etc. When used to model systems that run in an asynchronous environment, such programs are best viewed as non-deterministic, and we reject them for code generation. They are formally identified as programs that are not weakly endochronous [15, 16].

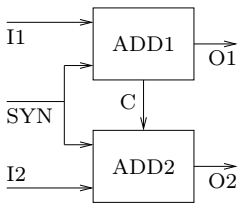
Instead of `MERGE`, we accept weakly endochronous specifications such as `WE_MERGE`, of Fig. 6, where the choice between one action or the other is given as the choice over the *value* of some input, and not the presence/absence of an input or the relative arriving order of two inputs.

We conclude with an example involving both concurrent and non-concurrent behaviors. The example, pictured in Fig. 7, models a simple reconfigurable adder, where two independent single-word ALUs (`ADD1` and `ADD2`) can be used either independently, or synchronized to form a double-word ALU. The inputs of each adder are modeled by a single signal (`I1` and `I2`). This poses no problem because from the synchronization perspective of this paper the two integer inputs of an adder have the same properties. At each execution cycle where `I1` is present, `ADD1` computes `O1` as the modulo sum of the components of `I1`.

```
process WE_MERGE =
(? int A,C ; bool X ; ! int O ;)
(| O = MERGE(A,C)
 | A ^= when X | C ^= when (not X) |) end


void onX() {
  bool X ;
  await_X(&X) ; if(X) onA() ; else onC() ;
}
void main() { for(;;) onX() ; }
```

FIG. 6 – Extending `MERGE` to a system that has a deterministic globally asynchronous implementation. Signal source (top) and event-driven implementation (bottom).



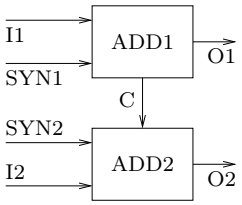

```
process ADDER =
(? int_pair I1,I2;
   event SYN;
 ! event O1,O2;)
(| (O1,C) = ADD1(SYN,I1)
 | O2 = ADD2(SYN,I2,C)
 | C ^= SYN
 | SYN ^< I1^*I2 |)
where event C; end
```

FIG. 7 – A reconfigurable adder that can function either as a single-word or a double-word adder.

Similarly, `ADD2` computes `O2` whenever `I2` is present. The choice between synchronized and non-synchronized mode is done using the `SYN` signal. When it is present, both `I1` and `I2` are present and a carry value is propagated from `ADD1` to `ADD2` through signal `C`. When `SYN` is absent, the adders function independently, and no carry is propagated, even if both adders are active during an execution cycle.

Obviously, the behavior of the specification depends on the presence or absence of signal `SYN`. Therefore, the specification is not weakly endochronous and cannot be given a globally asynchronous deterministic implementation. A weakly endochronous version of `ADDER` is provided in Fig. 8, where signal `SYN` is replaced by two signals `SYN1` and `SYN2` driving each one of the simple precision adders. Signal `SYN1` is present at each cycle where `ADD1` is present, and it has value `true` to specify that `ADD1` must synchronize with `ADD2` (by sending `C`). It has value `false` to specify that `ADD1` needs not to be synchronized.

A concurrent implementation of `WE_ADDER` is provided in Fig. 9. The following sections will explain how we determine that a program is weakly endochronous, and how we produce multi-threaded event-driven implementations.




```
process WE_ADDER =
(? int_pair I1,I2;
   boolean SYN1,SYN2;
 ! event O1,O2;)
(| (O1,O2) =
    ADDER(I1,I2,when SYN1)
 | when SYN1 ^= when SYN2
 | I1^=SYN1 | I2^=SYN2 |)
end
```

FIG. 8 – A weakly endochronous version of ADDER.

```
void ADD1() {
  bool SYN1,C; int O1 ; long I1 ;
  await_SYN1(&SYN1) ; await_I1(&I1) ;
  compute1(I1,&O1,&C);
  emit_O1(O1) ; if(SYN1) emit_C(C); }
void ADD2() {
  bool SYN2,C ; long I2 ;
  await_SYN2(&SYN2) ; await_I2(&I2) ;
  if(SYN2) {
    await_C(&C) ; emit_O(compute2(I2,C)) ;
  } else emit_O(compute2(I2,false)) ; }
void main() {/* driver pseudocode */
  start_thread{ for(;;) ADD1() ; } ;
  start_thread{ for(;;) ADD2() ; } ; }
```

FIG. 9 – Concurrent implementation of `WE_ADDER`.

2.1 Related work

The brute force compilation schemes of synchronous languages, together with several optimizations taking active modes and dynamic signal sensitivity into consideration, are accounted for in [3, 18] and further previous articles are mentioned in them. The optimized executions (here event-driven) go one (small) step further in the direction of asynchronous execution of synchronous programs.

The concern for distributed execution of synchronous programs goes back to Girault and Caspi [7]. While it looks at first presumably simple (mapping logical parallelism onto physical one), the reaction to absence mandates that void absent-signal messages are physically sent to whichever process/task contains the signal as input. This heavily penalizes the efficiency of the distributed code.

The notions of endochrony, and later weak endochrony, were introduced in [2, 11, 15]. They identify exactly the cases where two behavioral sets are independent, so that they can be executed concurently without disabling one another. A system can be made (weakly) endochronous by re-inserting the broadcast of absent signal messages, but only for those signals and execution cycles for which it is strictly needed (hopefully only a few cases for most specifications).

Although it deals with the signal values (which may be used to decide which further signals are to be present next causally in the reaction), endochrony is in essence strongly related with the notion of *conflict-freeness*, which simply states that once enabled, an event cannot be disabled unless it has been fired, and which was first introduced in the context of Petri Nets. Various conflict-free variants of data-flow declarative formalisms form the area of *process networks* (such as Kahn Process Networks [12]), or various so-called domains of the Ptolemy environment such as SDF Process Networks [5]. Conflict-freeness is also called *confluence* ("diamond property") in process algebra theory [13], and monotony in Kahn Process Networks.

Conflict-free process networks systems can then be made synchronous again by computing a *schedule* which assigns a precise instant for the activation condition to every operation. Various criteria may be used for constraint or optimization in this scheduling process. SoC *latency-insensitive design* [6, 4, 9] considers very specific hardware registers for buffer queues connecting the asynchronous tasks. N-synchronous processes [8] and polychronous (or multiclock synchronous) specifications [11] allow some freedom in the exact original timings, something between the strictly asynchronous and strictly synchronous case, and compute the schedule using so-called *clock calculus*.

Finally, the work presented in this paper is in strong relation to a number of previous papers of the authors (most recently [17]). In these papers, we develop the theory of weakly endochronous systems and abstract algorithms solving part of the considered implementation problem. The contribution of the current is to take these abstract notions and algorithms and make them effective, investigating code generation, distribution, language connection, and weak endochrony analysis issues with an efficiency-oriented viewpoint.

3 Weak endochrony

The theory of weakly endochronous (WE) systems [15] defines the sub-class of synchronous systems whose behavior does not depend on signal absence. WE synchronous programs represent systems whose behavior does not depend on the timing of the various computations and communications, or on the relative arrival order of two events. Weak endochrony is compositional and provides the necessary and sufficient condition ensuring a deterministic asynchronous execution of the specification. In particular, it determines that compound program reactions that are apparently synchronous can be split into independent smaller reactions that can be executed in any order, even at the same time, taking advantage of the potential concurrency of the specification. Most importantly, any possible program reaction can be generated by the union of such compound independent reactions.

In this section, we use the examples of Section 2 to introduce the weak endochrony variant on which our implementation technique is based, following the approach of [17].

Given a synchronous program P, we denote with $\mathcal{S}_P$ the set of all its variables (also called *signals*). All signals are typed, and we denote with $\mathcal{D}_S$ the data type of a signal S. A behavior (or *reaction*) of P is a tuple assigning one value to each of its signals. To represent cases where a signal can be absent, we use a special value denoted $\perp$. Thus, a reaction r of P assigns to each $S \in \mathcal{S}_P$ a value $r(S) \in \mathcal{D}_S^\perp = \mathcal{D}_S \cup \{\perp\}$.

Given a set of signals $\mathcal{S}$, we denote with $\mathcal{R}(\mathcal{S})$ the set of all valuations of the signals into their domains. We denote with $\mathcal{R}(P)$ the set of all reactions of a program P. Obviously, $\mathcal{R}(P) \subseteq \mathcal{R}(\mathcal{S}_P)$.

All possible reactions of the `SIMPLE` program of Fig. 2 are of one of the following forms :

$$\begin{aligned} r_1^v &= (\mathtt{A} = v, \mathtt{B} = f(v), \mathtt{C} = \perp, \mathtt{D} = \perp) \\ r_2^w &= (\mathtt{A} = \perp, \mathtt{B} = \perp, \mathtt{C} = w, \mathtt{D} = g(w)) \\ r_3 &= (\mathtt{A} = \perp, \mathtt{B} = \perp, \mathtt{C} = \perp, \mathtt{D} = \perp) \\ r_4^{v,w} &= (\mathtt{A} = v, \mathtt{B} = f(v), \mathtt{C} = w, \mathtt{D} = g(w)) \end{aligned}$$

where v and w range over $\mathbb{N}$. We have $\mathcal{R}(\mathtt{SIMPLE}) = \{r_1^v, r_2^w, r_3, r_4^{v,w} \mid v, w \in \mathbb{N}\}$.

We introduce on each $\mathcal{D}_S^\perp$ the partial order $\leq$ defined by $\perp \leq v$ for all $v \in \mathcal{D}_S$. This relation can be extended component-wise to a partial order on sets of reactions. For instance, $r_3 \leq r_1^v \leq r_4^{v,w}$ for all v, w. We denote with $\vee$ and $\wedge$ the least upper bound and greatest lower bound operators induced by $\leq$ on the sets $\mathcal{R}(\mathcal{S})$. For instance, $r_1^v \vee r_2^w = r_4^{v,w}$, and $r_1^v \wedge r_2^w = r_3$ for all v, w. We shall call the two operators *union*, respectively *intersection* of reactions. Since $\leq$ defines a lower semilattice, the $\vee$ operator is not always defined.

Whenever $r \vee r'$ is defined for two reactions r and r', we shall say that r and r' are non-contradictory and write $r \bowtie r'$. When this is not true, we write $r \not\bowtie r'$ and say that r and r' are contradictory. For instance, $r_1^v \bowtie r_2^w$ for all v and w, but $r_1^{v_1} \not\bowtie r_1^{v_2}$ for all $v_1 \neq v_2$. Obviously, r and r' are contradictory ($r \not\bowtie r'$) if and only if a signal S exists such that $r(S) \neq \perp$, $r'(S) \neq \perp$, and $r'(S) \neq r(S)$. This means that r and r' can be distinguished from one another by testing the

(present) value of S. This can be done even when observing the execution of P in an asynchronous environment where absence cannot be sensed. When $r \bowtie r'$ we also define their *difference* $r \setminus r'$ as the reaction exclusive with r' that satisfies $(r \setminus r') \vee (r \wedge r') = r$.

Definition 1 (Weak endochrony) *Under the previous notations, we say that a synchronous program P is weakly endochronous whenever $\mathcal{R}(P)$ is :*

- *Closed under $\vee$ (recall that $\vee$ is defined on $\mathcal{R}(\mathcal{S}_P)$).*
- *Closed under $\wedge$ and $\setminus$ applied on non-contradictory reactions.*

and it also includes the stuttering *reaction $\perp$ which assigns $\perp$ to all signals.*

From our code generation point of view, the most important consequence of this definition is that for any weakly endochronous program there exists a subset of reactions $Atoms(P) \subseteq \mathcal{R}(P) \setminus \{\perp\}$, called *atomic reactions* or *atoms*, with two key properties :

Generation : Any reaction $r \in \mathcal{R}(P)$ is uniquely defined as a union of zero or more atoms.

Independence : Any two different atoms that are non-contradictory share no common present signal, so that they can be freely united to form composed reactions.

In other words, atoms are the elementary reactions of P, and two atoms are either contradictory (they can be distinguished in an asynchronous environment), or independent (they can be executed without any synchronization).

The `SIMPLE`, `WE_MERGE`, and `WE_ADDER` examples are weakly endochronous. For instance, $Atoms(\texttt{SIMPLE}) = \{r_1^v, r_2^w \mid v, w \in \mathbb{N}\}$. The atoms r_1^v and r_2^w are independent for all v, w, and $r_1^{v_1}$ and $r_1^{v_2}$ are contradictory for all $v_1 \neq v_2$. The atoms of `WE_MERGE` (all the reactions different from $\perp$) are mutually contradictory. More interesting is the atom set of `WE_ADDER`. To represent it, we introduce the convention that absent signals are not represented in the tuple notation of a reaction. Then, the atoms are :

$$\begin{aligned} a_1^{i1} &= (SYN1 = f, I1 = i1, O1 = o1) \\ a_2^{i2} &= (SYN2 = f, I2 = i2, O2 = o2) \\ a_3^{i1,i2} &= (SYN1 = t, I1 = i1, O1 = o1, \\ &\qquad SYN2 = t, I2 = i2, O2 = o3, C = c) \end{aligned}$$

where t and f stand for *true* and *false*, $o1$ and c are the values produced by function `compute1` from $i1$, $o2 = \texttt{compute2}(i2, \mathit{false})$, and $o3 = \texttt{compute2}(i2, c)$. Note here that a_1^{i1} and a_2^{i2} are independent for all $i1$, $i2$, but they are both contradictory with $a_3^{i1,i2}$.

The last theoretical ingredient we need before presenting our code generation scheme is causality. The theory of WE systems is defined in a non-causal framework where no difference is made between input and output signals. To allow for the synthesis of systems that are deterministic, we need to ensure that the behavior only depends on input values. This property is easily expressed at the level of atoms, by requiring that any two conflicting atoms have one conflicting input :

Determinism : When two atoms a, a' are contradictory, there exists an input signal I that is present in both a and a' with $a(I) \neq a'(I)$.

```
void thread_a() {/* thread pseudocode */
  for( ; ;){
    await_inputs(I_1 = a(I_1),...,I_n = a(I_n)) ;
    for(i=1 ;i<=n ;i++)lock(I_i) ;
    compute() ;
    send_outputs() ;
    for(i=1 ;i<=n ;i++)
      consume_and_unlock(I_i) ;
  }
}
void main() {/* driver pseudocode */
  forall(a ∈ Atoms(P))
    start_thread{ thread_a() ; }
}
```

FIG. 10 – Simple generic implementation of a weakly endochronous program. In $\texttt{thread}_a$ the signals $\texttt{I}_1,\ldots,\texttt{I}_n$ are the inputs used by a.

4 Multi-threaded code generation

We explain in this section how concurrent multi-threaded, possibly distributed code can be generated from weakly endochronous programs. We shall explore some implementation variants and optimizations, including those leading to the implementations presented in Section 2.

The building blocks of any concurrent implementation are the atomic reactions of the program, which provide the elementary behaviors of the program that can run asynchronously from one another.

The simplest possible multi-threaded implementation of a synchronous program has one thread for each generator of the program, as pictured in Fig. 10. The thread `thread_a` corresponding to atom a cyclically performs the following sequence of operations :

1. Wait for the input configuration where the signals used as input by a have all arrived with the values specified by a.
2. Place a lock on these inputs. These locks inform other threads waiting for the same values on the same signals that the current values will be consumed by `thread_a`. The other threads will have to wait for new values.
3. Perform the actual computation and send the output signals.
4. Consume the current value, and then unlock each of the input signals. Consuming means removing the old value, and allows new ones to arrive so that other threads can advance.

The late consumption of the inputs, as opposed to creating a local copy in step 2 and allowing new inputs to arrive, ensures that fast computing atoms cannot overtake slow atoms, so that the outputs are well ordered and there is no conflict in writing the outputs. The lock mechanisms is part of the late consumption mechanism, forbidding a thread to use input values that are currently used and will be consumed by some other thread. Together, late consumption and locks ensure the atomicity of atom computation.

This form of multi-threaded implementation has two advantages (generality and simplicity), and three main drawbacks :

- The possible explosion in thread numbers (one per atom).
- The inefficiency of the late consumption and lock mechanism.
- Distribution problems determined by the fact that computation resources are used by threads built on semantic rather that resource locality arguments. In the `WE_ADDER` example, for instance, the `compute1` and `compute2` functions representing the computation of `ADD1` and `ADD2` are both used by $a_3^{i1,i2}$. If we seek a distributed implementation where `compute1` and `compute2` are placed on different processors, the computation of $a_3^{i1,i2}$ must be distributed itself.

We provide here solutions overcoming these drawbacks.

4.1 Reducing the number of threads

We present here two techniques for reducing the number of threads. They are both based on factoring common parts of several threads. Given the simple mapping between atoms and threads, these techniques can be formulated as techniques for the compact representation of atom sets, based on symbolic representation and hierarchization. As we shall see in Section 5, such compact representations of the atom sets support better (more efficient) analysis algorithms for verifying weak endochrony and constructing the atom set. In turn, using compact representations during analysis means that the analysis phase directly outputs compact representations of the atom set that can be directly encoded into efficient executable code.

A side-effect of the thread number reduction will be that the use of late consumption and locks is largely reduced.

4.1.1 Symbolic atom representation

Recall that atoms are reactions, which are valuations of the various signals. For instance, the atom definition a_1^{i1} of the previous section represents as many atoms as there are values in the domain of `I1`. When signals range over large or infinite domains, like the numeric types, generating one thread per atom is impossible. Even for small finite types, such as booleans, this translation is inefficient, as it artificially creates concurrency between threads representing exclusive computations, and thus raises the cost of synchronization.

The basic solution to this problem is provided by the very notation used to represent these atoms, which lets the value of input signals range over entire domains, like in the definitions of a_1^{i1}, where $i1$ ranges over all integer pairs that a single-precision adder can receive.

But replacing the inputs with variables ranging over domains also means that the outputs must be replaced with data expressions. For instance, in a_1^{i1} the value of `O1` is computed as one of the outputs of `compute1`$(i1, \&o1, \&c)$. In our example, this poses no problem. However, when the computed value is later used to make decisions (tests) that affect the synchronization (and therefore the number and form of the atoms), the formalism used to represent the atom sets must allow variables to range over inverse images of Boolean values through compositions of such data expressions. This also poses computability/complexity

problems at analysis time, as the analysis algorithms must compute these inverse images and then test their disjointness.

For these reasons, an exact analysis of weak endochrony is not feasible in the general case (any data type/function). In its current status, our prototype tool can analyze finite data types (`event`, `bool`, and enumerated types). All large and infinite types (including numeric ones) and all data functions on them are treated as uninterpreted types and function symbols.

The only function that accepts the symbolic representation presented above is identity. The resulting representation of an atom set is a set of symbolic atoms of the form :

$$< (S_1 \in D_1, \ldots, S_n \in D_n), Eq > \quad (1)$$

where a set of possible non-absent values D_i is specified for each signal S_i, and Eq is a set of equality relations of the form $S_i = S_j$ specifying that the values of the two signals must be identical.

All other data functions on finite types (*e.g.* the Boolean operators) can be analyzed if they are defined as sets of correct (*argument*, *result*) pairs that will be taken into account like any other atom set. All other data functions are treated as uninterpreted function symbols.

The result is an abstraction similar to the finite stateless abstraction used by the Signal/Polychrony compiler [14]. The expressivity is satisfactory, as it allows the representation and analysis of all combinational (non-sequential) clock relations used in the compilation of synchronous languages, like the clock tree of Signal/Polychrony[1] or the selection tree of Esterel[18]. The abstraction is conservative : Some weakly endochronous programs will be rejected for code generation, but all accepted programs are weakly endochronous. The analysis algorithm presented in the next section is exact when no type or function is uninterpreted.

Starting from this symbolic representation, code generation is done using a simple variant of the generic technique defined above. For each symbolic atom a of the form given in Equation 1 one thread $\texttt{thread}_a$ is produced. The only difference between this thread and the one of Fig. 10 is that the `await_inputs` statement is replaced with a more complex one capable of detecting input configurations belonging to a symbolic atom :

`await_inputs(`$\texttt{I}_1 \in \texttt{D}_1$`,...,`$\texttt{I}_n \in \texttt{D}_n$`,`$Eq$`) ;`

where D_i is the set of possible non-absent values specified by a for I_i and Eq specifies which equality relations must hold between the input signals.

The code generated for example `SIMPLE` in Fig. 4 is generated using this technique, whereas the non-symbolic technique could not handle it.

4.1.2 Hierarchic atom set representation

The generic code generation technique defined above produces code containing no conditional (`if`) statement. Thus, even when two atoms correspond to the branches of a test we still have to use two threads to encode them, with the accompanying synchronization overhead. For instance, this is the case for atoms a_1^v and $a_3^{v,w}$ of `WE_ADDER`, which correspond to the branches of a test on the Boolean value of `SYN1`.

This is clearly suboptimal. Our objective in this section is to allow the grouping of atoms into sets of mutually contradictory atoms so that each set can be

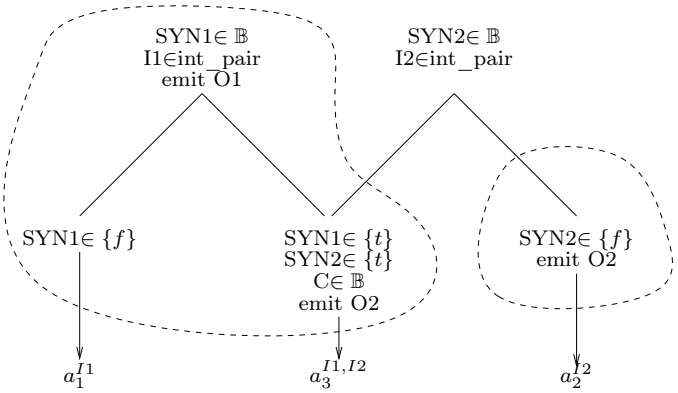


FIG. 11 – Hierarchic atom set representation and partition into threads

encoded by one single sequential thread. By sequential thread we understand here code formed using `if` tests and the primitives already used by the generic implementation and its symbolic extension. Not all sets of mutually contradictory atoms can be transformed into a single thread. The simplest example where this is impossible is the following system having 3 Boolean inputs (`A`, `B`, and `C`) and the atoms :

$$(A = t, B = t), \quad (B = f, C = t), \quad (A = f, C = f)$$

Encoding into sequential code is impossible here because no static order of wait and test statements allows the choice between the 3 atoms.

However, when such a static order exists, we represent it using a data structure similar to the *endochronous clock trees* used in the compilation of Signal/Polychrony[1, 2]. The hierarchic data structure is a decision tree, which we exemplify on the left tree of Fig. 11 (the tree rooted in $SYN1 \in \mathbb{B}$ and surrounded by a dashed line). Each node of the tree contains a symbolic input configuration similar to the input of `await_signal` and a set of actions to be realized (signal emissions and/or function computations). For the tree to be a decision tree, the input configurations of two nodes having the same parent node must be contradictory, and the union of the configurations of the children nodes must cover the parent node. For compactness reasons, the input configuration of a node can be incomplete. The represented domains and equality constraints will be seen as supplementary constraints over the ones of the parent. Each leave of the tree corresponds to exactly one atom. To ensure coherence with the late consumption mechanism, all the atoms whose input configuration is smaller than the input configuration of a node n must be represented by leaves of the sub-tree rooted in n. In the example tree, the top node specifies reactions where `SYN1` and `I1` are available and `O1` is emitted. Its two children represent the symbolic atoms a_1^{i1} and $a_3^{i1,i2}$ which correspond to the branches of a test on `SYN1`.

Generating code from such a tree consists in transforming the hierarchy of the tree into a hierarchy of data tests. For each input configuration in the tree :

– If the configuration includes signals that are not present in the configurations of higher-level nodes, these signals become the arguments of an `await_input` statement.

– If the configuration includes signals that are present in the configurations of higher-level nodes (which should be the case for all nodes but the root one), they are transformed into the test conditions of the data tests.

When an input signal is only used by one thread, no lock is needed for it, because the cyclic execution of the thread ensures the needed exclusiveness property. The code generated for the left thread of Fig. 11, provided below, needs no lock on signals SYN1 and I1.

```
void thread_1() {/* thread pseudocode */
  for( ; ;){
    await_inputs(SYN1,I1) ; lock(SYN1,I1) ;
    compute1(I1_val,&c,&o1) ;
    send_output_O1(o1) ;
    if(SYN1_val){
      await_inputs(SYN2∈ {t},I2) ;
      lock(SYN2,I2) ;
      send_output_O2(compute2(I2_val,c)) ;
      consume_and_unlock(SYN2,I2) ;
    } } }
```

The data structure defined above not only allows the representation of decision trees associated with threads, but also the representation of full atom sets, as exemplified in Fig. 11 (the full picture). Instead of a tree, the representation is a forest with possibly multiple toplevel nodes. To generate code from such a forest, we first determine a subset of nodes of the forest such as as the subtrees rooted in these nodes define a partition of the leaves (symbolic atoms). In our example, two nodes/trees are necessary. Then, code generation is performed separately for each tree. The code for the second thread of the adder is :

```
void thread_2() {/* thread pseudocode */
  for( ; ;){
    await_inputs(SYN2∈ {f},I2) ;
    lock(SYN2,I2) ;
    send_output_O2(compute2(I2_val,0)) ;
    consume_and_unlock(SYN2,I2) ;
  } }
```

The remaining question is that of organizing flat atom sets into decision forests. The good news is that any amount of hierarchization is good, and the technique should work even on flat symbolic atom sets. One solution is to reconstruct hierarchy from flat atom sets at code generation time. Previous work of Talpin *et al.* [19] produces decision forests for particular cases of weakly endochronous programs. We are currently investigating the use of forest representations of the atom set to improve the efficiency of weak endochrony analysis algorithms. Results in this direction will provide us with hierarchic representations without the need of re-constructing them from flat representations.

4.2 Distributed code generation

In the previous sections we focused on generating multi-threaded code, but without considering distribution problems arising from resource locality. Assume, for instance, that the code of the WE_ADDER example must be distributed

over two processors, one responsible for receiving `I1`, emitting `O1`, and performing the computation of `ADD1`, and the other responsible for receiving `I2`, emitting `O2`, and performing the computation of `ADD2`. Then, the code produced by the technique of the previous section is not good, because `thread_1` involves signals and computations of both `ADD1` and `ADD2`.

As each thread corresponds to an endochronous decoding process, the first idea would be to rely on previous work by Girault and Caspi [7], applied separately on each thread. However, this approach does not work due to the particular semantics of the `await_inputs` statement, which atomically waits for multiple input configurations instead of the simple arrival of individual signals, and considers the inputs read only when the configuration is complete. In single-processor implementations this primitive can easily be implemented by input polling. However, in a multi-processor framework where each input arrives to exactly one processor, direct polling is impossible. Some data must be explicitly exchanged between processors to allow computation to advance, which amounts to replacing the polling-based protocol with one based on classical single-input blocking `wait` statements which do not involve domain tests.

This transformation amounts to providing an implementation of the atomic `await_inputs` over blocking `wait`s. This transformation is necessarily global, because a signal can be argument to several `await_inputs` statements. Since one blocking wait is generated for each `await_inputs` statement, the input must be broadcast to all active wait points.

Once this expansion has been realized, a technique similar to that of OCRep can be applied : The hierarchic atom set representation is replicated on every processor in the system, and then simplified by removing all unnecessary operations and signals. Then, the simplified forest of each processor is independently implemented. The result of this process for our `WE_ADDER` example is presented in Fig. 9.

5 Weak endochrony analysis

In [17], a set of algorithms has been defined for checking if a synchronous process is weakly endochronous. The algorithms are based on the construction of so-called *generator sets*, which have generation and independence properties similar to those of an atom set, but can be defined on any synchronous program. The construction is performed inductively (bottom-up) on the structure of the synchronous program, and the program is weakly endochronous whenever the generator set has the properties of an atom set. We first review the approach, and then explain how we adapted it to suport our new code generation techniques.

The key concepts here are generators and generator sets. Their definition was based on the simple observation that atom sets cannot be directly used to represent the behavior of general synchronous programs. The simplest example of this is the `MERGE` program of Fig. 5. Its elementary reactions are the same as the r_1^v and r_2^w atoms of `SIMPLE`, defined in Section 3. However, in `MERGE` these two reactions are not independent, and cannot be united, because signals `A` and `C` are exclusive.

Thus, an exclusiveness relation exists between the elementary behaviors of `MERGE` which is not implied by the conflict of some present signal values, but by the presence/absence of a signal. To represent this exclusiveness relation

between elementary behaviors, we introduce a special notation $\perp\!\!\!\perp$ representing *forced absence*. Intuitively, a signal S is set to $\perp\!\!\!\perp$ top denote the fact that the present valuations of the other signals determine the absence of S. Under this notation, the two basic behaviors of `MERGE` can be represented by the *generator set* :

$$\begin{array}{ll}\{ & (A = v, B = f(v), C = \perp\!\!\!\perp, D = \perp\!\!\!\perp) \\ & (A = \perp\!\!\!\perp, B = \perp\!\!\!\perp, C = v, D = g(v)) \quad \}\end{array}$$

For instance, the first generator specifies that once `A` is present, `C` and `D` are absent.

However, this generator set contains more information than needed, as any one of the four $\perp\!\!\!\perp$ values represent the same exclusiveness information. Since more $\perp\!\!\!\perp$ values directly translate into longer analysis time and more complex generated code, we will prefer generator sets that minimize the number of $\perp\!\!\!\perp$ values, such as :

$$\begin{array}{ll}\{ & (A = v, B = f(v), C = \perp, D = \perp) \\ & (A = \perp\!\!\!\perp, B = \perp, C = v, D = g(v)) \quad \}\end{array}$$

Such reactions where signals can take a $\perp\!\!\!\perp$ value are called *extended reactions*. The order relation on (normal) reactions is extended to one on extended reactions by assuming that $\perp \leq \perp\!\!\!\perp$. Thus, the new value is treated as any present value for reaction ordering, intersection, unification, and the non-contradiction relation on extended reactions.

Under these notations, a set of extended reactions G of a program P is a generator set for P if is satisfies the two key properties of an atom set :

– Unique **generation** of all reactions of P

– **Independence** between generators sharing no common present signal.

Note that the independence property is the only place where $\perp\!\!\!\perp$ is not treated as a present value. This is necessary to preserve the concurrency of the representation. For instance, a generator set of the non-endochronous `ADDER` example of Fig. 7 is composed of :

$$\begin{array}{rcl} g_1^{i1} & = & (I1 = i1, O1 = o1, SYN = \perp\!\!\!\perp) \\ g_2^{i2} & = & (I2 = i2, O2 = o2, SYN = \perp\!\!\!\perp) \\ g_3^{i1,i2} & = & (I1 = i1, O1 = o1, \\ & & \quad I2 = i2, O2 = o3, SYN = \bullet, C = c) \end{array}$$

where $SYN = \bullet$ states that `SYN` is present, and the values $o1$, $o2$, $o3$, and c are as defined in Section 3 for example `WE_ADDER`. Generators g_1^{i1} and g_2^{i2} are independent and can be freely composed, but they are both exclusive with $g_3^{i1,i2}$.

Based on this theory, we have previously defined [17] algorithms able to construct a minimal generator set of a synchronous program. This construction is fundamental, because the program is weakly endochronous whenever this minimal generator set contains no $\perp\!\!\!\perp$ signal valuation, in which case it is the atom set of the program and we can apply on it the code generation of Section 4. However, these algorithms are defined on simple data structures without symbolic and hierarchic generator representation. These data structures have the same problems as flat atom sets, as explained in Section 4. To allow an efficient

Example	Signals (In/Out/Loc)	Statements	Running time	Generators
orcilloscope model	9/9/45	78	17.7s	89
mouse handler	4/9/35	61	3.4s	34
equation solving	8/5/34	53	10.7s	48

TAB. 1 – Partial experimental results

analysis of weak endochrony, we defined a symbolic generator set representation and defined new algorithms able to exploit such a representation. The difficulties come from the fact that each symbolic generator represents a set of generators. Thus, for each test in the old version of the algorithms we now have to partition the input generator set into a subset taking the `then` branch and the subset taking the `else` branch. This form of partitioning is cumbersome, because it not necessarily results in two symbolic generators (each generator set describes a convex set of generators, but the difference between two convex sets is not necessarily convex).

We are currently working on extending the analysis algorithms to work on hierarchic generator set representations. The difficulty here is that of composing different hierarchies with different signal test orders.

6 Preliminary results

The implementation of the full code generation tool is not yet completed : We have completed a first version of the endochrony analysis part, but not the generation of multi-threaded code.

However, intermediate results are already available shedding some light on the form and efficiency of the final code. The results are listed in Table 1.

We used 3 representative Signal/Polychrony examples of average size. The running times of our implementation (of the order of seconds on an Intel Core2 Duo CPU running at 2.8GHz) are encouraging, given that we mostly focused our work on the data structures, so that much remains to be done in the algorithm optimization field. Memory was not an issue, even for larger examples.

The number of generators corresponds to the symbolic representation of Section 4.1.1 which uses signal domains, but no interpreted function, and no hierarchic compaction. The figures are still large, given that one thread is created for each generator. However, preliminary tests on the hierarchic representation of generator sets drastically reduce these figures (2 threads for the equation solving example, 5 for the oscilloscope example). Moreover, full expansion is not without interest, as for small and medium size programs it may allow better optimization of the code for each generator.

7 Conclusion

We have fully defined in this paper the first general technique for the multi-threaded, possibly distributed asynchronous implementation of polychronous synchronous speficications, going all the way from high-level specification to deterministic generated code.

Starting from the theory of weakly endochronous systems [15] and from the abstract analysis algorithms of [17], we have provided practical algorithms, along

with efficient symbolic data structures allowing the efficient analysis of weak endochrony and the generation of well-structured multi-threaded implementation code.

We have implemented our symbolic data structures and algorithms in a prototype version, linked to the Signal/Polychrony-SME design environment as a back-end tool. Thus, programs are written in Signal, and various other analysis available for this language are thus possibly combined with our distributed code generation. Benchmarking on existing large Signal programs is under way.

We currently focus on the extension of the data structures and analysis algorithms to use hierarchical generator representations. Other future work directions include a complexity analysis of the approach (in addition to benchmark results), and improving distributed code generation.

Références

[1] P. Amagbégnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Proceedings PLDI'95*, La Jolla, CA, USA, June 1995.

[2] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages : Specification and distributed code generation. *Information and Computation*, 163 :125 – 171, 2000.

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1) :64–83, January 2003.

[4] J. Boucaron, R. de Simone, and J.-V. Millo. Latency-insensitive design and central repetitive scheduling. In *Proceedings MEMOCODE'06*, Napa, CA, USA, 2006.

[5] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogenous systems. *International Journal in Computer Simulation*, 4(2), 1994.

[6] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9) :18, Sep 2001.

[7] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3) :416–427, May/June 1999.

[8] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks : a relaxed model of synchrony for real-time systems. In *Proceedings POPL'06*, pages 180–193. ACM Press, 2006.

[9] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. Desynchronization : Synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10) :1904–1921, October 2006.

[10] S. Dasgupta, D. Potop-Butucaru, B. Caillaud, and A. Yakovlev. Moving from weakly endochronous systems to delay-insensitive circuits. *Electronic Notes in Theoretical Computer Science*, 146(2) :81 – 103, 2006. Proceedings

of the Second Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS 2005).

[11] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, April 2003. Special Issue on Application Specific Hardware Design.

[12] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing '74*, pages 471–475. North Holland, 1974.

[13] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[14] M. Nebut. Specification and analysis of synchronous reactions. *Formal Aspects of Computing*, 16(3) :263–291, august 2004.

[15] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2) :111–130, March 2006.

[16] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Deterministic execution of synchronous programs in an asynchronous environment. a compositional necessary and sufficient condition. Research Report RR-6656, INRIA, 2008.

[17] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J.-P. Talpin. From concurrent multiclock programs to deterministic asynchronous implementations. In *Proceedings of the 9th Application of Concurrency to System Design Conference, ACSD'09*, Augsburg, Germany, July 2009.

[18] D. Potop-Butucaru, S. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[19] J.-P. Talpin, J. Ouy, T. Gautier, L. Besnard, and P. Le Guernic. Compositional design of isochronous systems. *Science of Computer Programming*, 2010.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399